

Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller

Nicolas Moro^{*†}, Amine Dehbaoui[†], Karine Heydemann[‡], Bruno Robisson^{*}, Emmanuelle Encrenaz[‡]

^{*}Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA)
Gardanne, France

Email: {nicolas.moro, bruno.robisson}@cea.fr

[†]École Nationale Supérieure des Mines de Saint-Étienne (ENSM.SE)
Gardanne, France

Email: amine.dehbaoui@mines-stetienne.fr

[‡]Laboratoire d'Informatique de Paris 6 (LIP6)

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6
Paris, France

Email: {nicolas.moro, karine.heydemann, emmanuelle.encrenaz}@lip6.fr

Abstract—Injection of transient faults as a way to attack cryptographic implementations has been largely studied in the last decade. Several attacks that use electromagnetic fault injection against hardware or software architectures have already been presented. On microcontrollers, electromagnetic fault injection has mostly been seen as a way to skip assembly instructions or subroutine calls. However, to the best of our knowledge, no precise study about the impact of an electromagnetic glitch fault injection on a microcontroller has been proposed yet. The aim of this paper is twofold: providing a more in-depth study of the effects of electromagnetic glitch fault injection on a state-of-the-art microcontroller and building an associated register-transfer level fault model.

Keywords—microcontroller, timing fault, electromagnetic glitch, fault attack, fault model

I. INTRODUCTION

Physical attacks aim at breaking cryptosystems by gaining information from their implementation instead of using theoretical weaknesses. Those attack schemes were introduced in the late 1990s. There are two main subclasses of physical attacks: passive and active ones. In passive attacks, an attacker uses the fact that some measurable data may leak information about manipulated secret data such as cryptographic keys. Physical quantities which can be used for passive attacks include execution time [1], electromagnetic radiations [2], power consumption [3] or light emissions [4]. In active attacks, an attacker modifies the circuit's behaviour in order to perform its attack scheme. Fault attacks are a subset of active attacks in which an attacker injects a transient fault in a circuit's computation.

Faults attacks were introduced in 1997 by Boneh *et al.* [5]. They consist in modifying a circuit environment in order to

change its behaviour or to induce faults into its computations [6] [7] [8]. Many means are of common use to inject such faults, especially laser shots [9] [10] [11], overclocking [12] [13], chip underpowering [14] [15], temperature increase [16] or electromagnetic glitches [10] [17].

There are three main subclasses of fault attacks: algorithm modifications, safe error and differential fault analysis. Algorithm modifications aim at skipping [18] or replacing [13] some instructions executed by a microcontroller to circumvent its security features. Safe-error attacks aim at evaluating whether or not a fault injection has an impact on the output [19]. Differential fault analysis (DFA) aims at retrieving the keys used by an encryption algorithm by comparing correct ciphertext and faulty ciphertexts (i.e. ciphertexts obtained from a faulted encryption). This technique was first introduced for public key encryption algorithms [5], and quickly extended to secret key algorithms [20].

From that time, many attack schemes have been proposed to attack various encryption algorithms. They all rely on an attacker's fault model which defines the type of faults the attacker can perform [21]. Thus, they require a high accuracy in the fault injection process. If the faults are not induced at the proper time in the algorithm, or affect the wrong bits, the entire attack process fails. As a consequence, the ability to precisely control the fault injection process is a key element in carrying out any fault attack. Common fault models include instruction skips [18], single bit faults or single word faults [22].

In this work, we report the use of electromagnetic pulses to induce faults into the computations of an up-to-date microcontroller. We also report a study of the local effect of electromagnetic pulses. Moreover, the underlying effects behind common fault models are not always clearly understood and may highly depend on the target architecture. As a consequence, this work finally aims at defining a precise fault model and providing an understanding of the faults an electromagnetic glitch can induce on an embedded program.

The rest of this paper is organized as follows. Section II introduces our fault injection experimental setup and details the approach we use. Section III describes the influence of some experimental parameters on injected faults. Section IV details the effects of the injected faults on the program flow and data flow. Finally, the resulting register-transfer level fault model is presented in section V. Section VI gives details about some related research papers.

II. APPROACH

This section starts by describing our experimental setup choices in II-A. This experimental setup enables us to provide the results presented in section III which show the influence of the different experimental parameters. Then, we detail the approach we use to precisely characterize the faults we injected. This characterization method, which matches experimental results obtained from the microcontroller with simulation data, is detailed in II-B.

A. Experimental setup

1) *Electromagnetic fault injection bench:* The electromagnetic glitch fault injection platform shown in Fig. 1 is composed of a control computer, the target device, a motorized stage, a pulse generator, and a magnetic antenna. The target (described in II-A2) is mounted on the X Y Z motorized stage. The computer controls both the pulse generator (through a RS-232 link) and the target board (through a USB link).

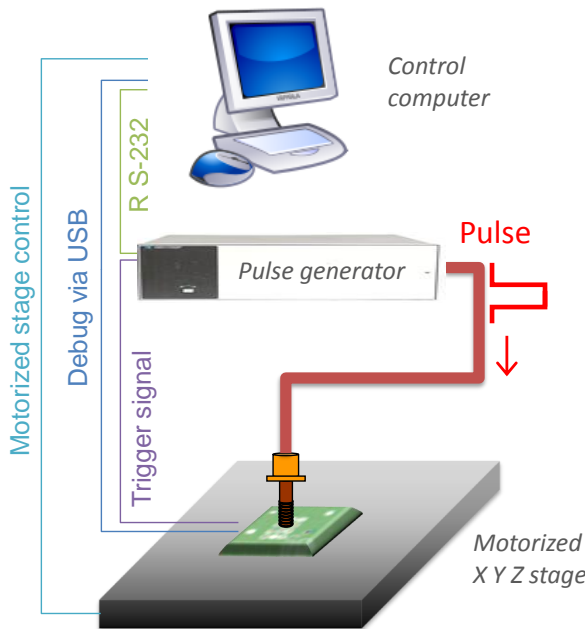


Figure 1: Electromagnetic fault injection bench

The pulse generator is used to deliver voltage pulses to the magnetic coil. It has a constant rise and fall transition time

of 2 ns. The amplitude range of the generated pulses extends from -200 V to 200 V, their width extends from 10 ns to 200 ns. The magnetic antenna we use is composed of a few turns with a diameter of 1 mm. We use it in order to disturb a small part of the target device. This spatial accuracy is possible thanks to a high accuracy X Y Z stage.

2) *Target:* The chosen target is an up-to-date 32-bit microcontroller designed in a CMOS 130 nm technology. It is based on the ARM Cortex-M3 processor [23]. Its operating frequency is set to 56 MHz. This microcontroller does not embed any cache memory.

Choice of target: The target we use is a state-of-the-art microchip, based on a recent technology. ARM Cortex processors are already very widespread for both mainstream and secure microcontrollers. Although we did not choose a smartcard version of the microcontroller, this target embeds some basic security mechanisms against clock perturbations and voltage glitches. Moreover, several interrupt vectors have been defined which can handle some hardware faults and can be used for a basic fault detection. Hence, we can consider this target as reasonably secured against some of the most common low-cost fault injection means. However, this target does not embed any protective shield against reverse engineering or electromagnetic injection. Since this research aims at understanding the effects of fault injection on a recent microcontroller, we do not work on a highly-secure version of this microcontroller.

Instruction set: Cortex-M3 processors run the ARM Thumb2 instruction set [24]. Thumb2 is actually the successor to both ARM and Thumb instruction sets, and contains both 16-bit and 32-bit instructions.

Hardware interrupts: Several fault exceptions can catch illegal memory accesses or illegal program behaviour. Those fault exceptions are Hard Fault, Bus Fault, Usage Fault and Memory Management Fault. Each of these exceptions can be triggered for several subtypes of hardware faults. In the following experiments, every exception handler function executes an infinite loop.

B. Experimental process

Working with a microcontroller in such a black-box approach requires to develop a specific experimental approach. This approach aims at enabling us to deduce the effects of faults by observing some internal data from the microcontroller. This observation must be done with a non-invasive technique. Since a faults may have an impact on the program flow and since we need to access some accurate data such as registers or cycle count, the communication cannot be done with a serial link. We use the JTAG-equivalent non-intrusive SWD debug link to retrieve data from the microcontroller. Besides, we also use the hardware exceptions defined in II-A2 as a way to get some extra data about the injected faults.

1) *Microcontroller's internal state observation:* The experimental measurement process we use is the following:

- Reset the microcontroller
- Execute the target code
- Send a pulse to the injection antenna
- Interrupt the program execution
- Harvest the microcontroller's internal data

The following paragraphs detail the important elements of this experimental process.

Trigger window: In order to have a correct view of the microcontroller's internal data, we have created an assembly subroutine containing some test instructions (which will be detailed in section III). For our experiments, the microcontroller sets a trigger signal for the electromagnetic injection. With this technique, we can target the executed program at the scale of a single instruction. By observing the microcontroller's clock during this trigger window, we can focus the injection on a single clock cycle. Besides, the pulse injection time is defined by reference to the beginning of the trigger signal temporal window.

Watchpoint and program end: In this experimental process, the program normally stops because of a breakpoint set after the target code. This watchpoint is defined before popping the stack at the end of the assembly subroutine and after the trigger window. However, with our experimental setup and target code, two other scenarios may happen because of a fault: an unconditional jump and an infinite loop due to the triggering of an exception. These two scenarios modify the control flow, and the program may not reach the defined breakpoint. Moreover, the unconditional jump scenario makes the setting of breakpoints very hard. To handle these issues, our control computer stops the microcontroller after a fixed delay.

Internal data: With the SWD debug link, the internal data we get from the microcontroller at a watchpoint for our experiments are: the general-purpose registers ($r0$ to $r12$), the stack pointer ($r13$), the link register ($r14$), the program counter ($r15$), the program status register ($xPSR$), some chosen variables in memory and the number of clock cycles taken by our experiment. This number of clock cycles is counted from the beginning of the target subroutine. The $xPSR$ register gives us information about the processor flags and the exceptions that may have been triggered. Since we only inject transient faults and since the watchpoint is set several clock cycles after our attack, we can reasonably assume that the debugging module embedded in the chip is not corrupted when recovering the internal data from the microcontroller.

However, some internal data such as the instruction register cannot be accessed. When working at the scale of a single instruction, we may need to determine which instruction has been actually executed by the core. To get a list of suitable instructions, we need to rely on an exhaustive instruction simulation.

2) *Fault model simulation:* We propose to use simulation to explain the effects of electromagnetic fault injection. Our approach aims at comparing the experimental faulty outputs with outputs from a fault model simulation. Thus, we can validate the interpretation of these effects by comparing the outputs with the internal data. This scheme is summarized in Fig. 2.

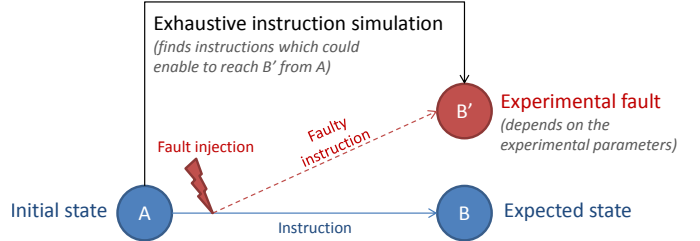


Figure 2: Our approach to characterize the injected faults

Simulations aim at finding output states which could be compatible with the output states we observed. In order to match simulations with measurements, we define a binary relation between experimental output states and simulated output states.

Definition. One instruction replacement **can explain** an experimental measurement if the output states ($[r0-r12]$, $xPSR$) at the defined watchpoint are the same for the measurement and the simulation

In the rest of this article, two classes of faults can be distinguished: faults on the *data flow* and faults on the *program flow* [22]. Faults that lead to the replacement of an instruction by another one are faults on the program flow. They may result in an algorithm modification, depending on the context and the replaced instruction. On the contrary, faults which only modify a piece of data without modifying an instruction are faults on the data flow.

Nevertheless, this difference might not be clear for many cases since both fault classes may lead to very similar visible outputs. Thus, defining whether a resulting faulty output is a consequence of a fault on the data flow or the control flow is generally a tough task. Nevertheless, a single assembly instruction can only output a very limited set of data. As a consequence, it is possible to tell whether or not a faulty output is the consequence of a fault on the control flow. Thus, every faulty output which cannot be explained by an instruction replacement is considered to come from a fault on the data flow.

The Thumb2 instruction set is composed of both 16-bit and 32-bit instructions. 16-bit instructions can be exhaustively tested. 32-bit instruction start with the prefixes 11101 or 1111, which reduces the complexity of an exhaustive test. Moreover, the 32-bit part of the instruction set is mostly sparse, we can remove many branches in the search space.

It should be noted that this simulation is performed on the

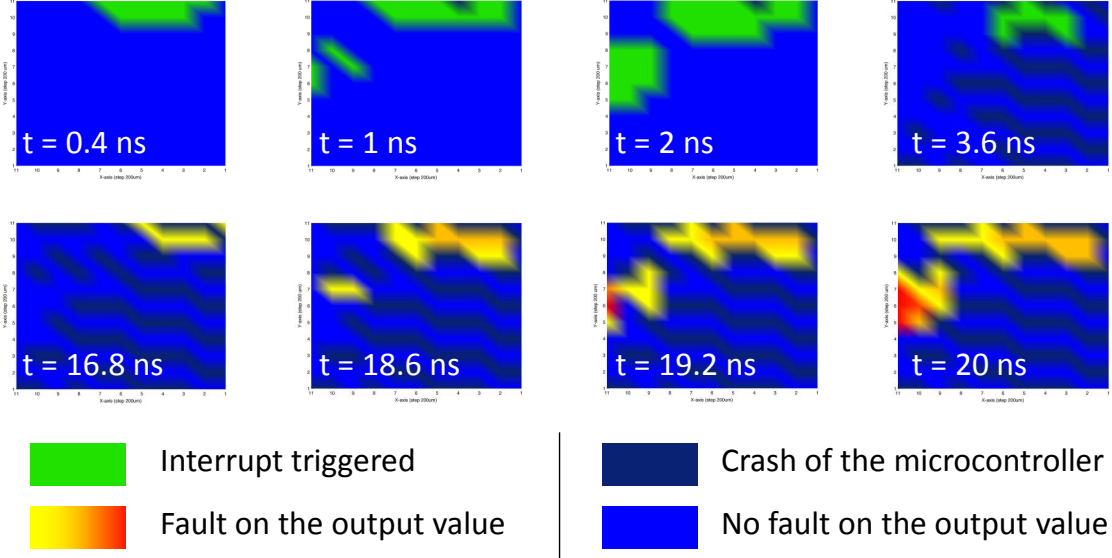


Figure 3: Impact of the probe's position

same binary as the one that is used for the fault injection experiments. To perform this simulation, we developed a specific program, based on the Keil UVSOCK library. Our simulation program is able to control the Keil μ Vision debugger during an execution on the Keil μ Vision simulator. It emulates faults on the control flow by replacing on the fly the target instruction.

Obviously, many instruction replacements may be able to explain one single measurement. Nevertheless, being able to simulate instruction replacement will enable us to explain the effects we observed and then to define a fault model more clearly. To sum up, an exhaustive simulation over the instruction set is practical and can be performed in real conditions. Moreover, it enables to distinguish faults on the control flow from faults on the data flow.

III. EXPERIMENTAL STUDY OF THE INJECTION PARAMETERS

In this section, we provide a study of the influence of several experimental parameters on the final outputs. Since metastability phenomena appear, we first start by describing them in the following paragraph.

A. Metastability phenomena

Since electromagnetic glitch fault injection leads to timing faults [17], we obtained some metastability phenomena. For this experiment, the pulse's voltage was set to 190 V, the clock frequency was set to 56 MHz, the pulse's injection time was fixed to an arbitrary value, and the pulse width was set to 10 ns. The probe position was found by a trial-and-reset approach. The results for 10000 executions of our experimental process are presented in Table I, every observed output value is associated to its occurrence rate.

They show a metastability phenomenon for a single load instruction from the Flash memory which correct loaded value is 0×12345678 since several values appear for the same fixed configuration of the experimental parameters.

Table I: Metastability phenomenon for a single load instruction

Loaded value	Occurrence rate
1234 5678 (no fault)	60.1%
FFF4 5679	27.4%
FFFC 5679	12.3%
FFFC 567b	0.1%
FFFC 7679	0.1%

B. Study of the injection parameters

In the case of an electromagnetic fault injection on a microcontroller, many experimental parameters can have an influence on the final outputs. The main parameters we can control in these experiments and which may have an influence are detailed in Table II. For all the following experiments except the one that studies the voltage's influence, the pulse voltage was set to 190 V. The pulse width was set to 10 ns, which is shorter than the 17 ns clock period (for a 56 MHz clock frequency). In the following paragraphs, we detail the separate influence of some of these parameters.

Table II: Experimental parameters

Electromagnetic injection parameters	- x-y-z position of the injection probe - Pulse injection time - Pulse characteristics (width, voltage)
Microcontroller hardware parameters	- Operating frequency - Power supply
Microcontroller software parameters	- Type of the executed instructions - Program memory (RAM or Flash)

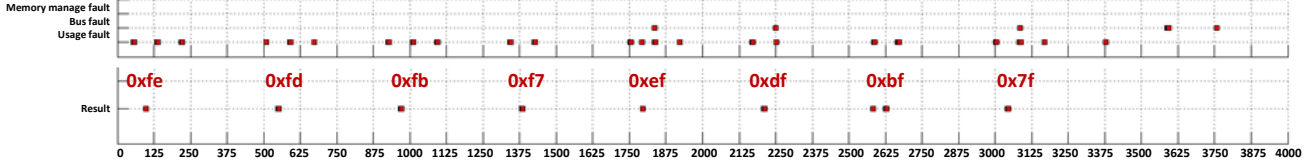


Figure 4: Influence of the pulse's injection time for an array sum whose expected result is 0xFF

1) *Position of the injection probe over the package's surface:* The X Y Z stage we use for our experiments enables us to vary the injection probe's position. Since varying the Z position of the antenna leads to a similar class of effects on the microcontroller than varying the pulse's voltage [25], we fix a position for Z and only study the influence of the X Y position. In this experiment, we change the X Y coordinates and the pulse's injection time. This experiment is performed at the scale of a single load instruction which loads the value 0x12345678 from the Flash memory into the register R8. This fault injection has been performed over a 20 ns time interval, by steps of 200 ps. The probe browsed a 3 mm square over the circuit's die, by steps of 200 μ m. Fig. 3 shows the results for this experiment.

The experiment shows that there are four kinds of outputs, depending on the probe position and the injection time : no fault on the loaded value, a crash of the microcontroller, the triggering of a Usage Fault exception, and a fault on the value in R8. Very few faults on the register R0 were also observed. Except from R8 and R0, no other register was faulted in this experiment. Moreover, those two registers were never faulted together. Every faulty output we observed on R8 has a higher Hamming weight than the 0x12345678 expected value. On Fig. 3, yellow areas led to a small increase in this Hamming weight and red areas led to a high increase. This experiment highlights the local effect of electromagnetic fault injection on a microcontroller, with different effects depending on the probe's position. Since very few probe positions can lead to a successful fault injection, this spatial cartography also helps us to find some suitable X Y Z configurations for the following experiments.

2) *Injection time:* This experiment has been performed on the following test program for a fixed X Y Z position. This program uses a loop to sum the elements of an array that contains eight powers of two. `array[i]` contains 2^i . At the end of the computation, the result stored at the address pointed by `r0` contains 0xFF. This test program requires about 3.5 μ s to complete. We performed this fault injection over this time interval, by steps of 200 ps.

```

1 addition_loop:
2   ldr r4, [r2, r1, lsl #2] ; r4 = array[i]
3   ldr r3, [r0, #0]        ; r3 = result
4   add r3, r4               ; r3 = r3 + r4

```

```

5   str r3, [r0, #0]        ; result = r3
6   add r1, r1, #1          ; r1 = r1 + 1
7   cmp r1, #8              ; r1 == 8 ?
8   blt addition_loop

```

This test program enables us to perform an electro-magnetic fault injection on a sample made of different instructions. The results for this experiment are shown in Fig. 4. Three kinds of situations have been observed:

- BusFault or UsageFault hardware interrupts
- A fault on the output value
- A normal behaviour with no fault

Every fault we observed on the output value corresponds to an execution in which only one power of two has not been added. However, many faults could explain such results. That is why the precise effect of electromagnetic fault injection at the scale of a single instruction is studied precisely in section IV.

3) *Pulse characteristics:* In the following paragraphs, we study the separate influence of the pulse parameters. For these paragraphs, a fixed position was set for the injection probe. This position had been found thanks to the spatial cartography presented in III-B1.

Pulse width: The pulse width does have an influence on the outputs. According to Faraday's law of induction, the electromotive force induced in a loop (e.g. inside the power grid) corresponds to the time-derivative of the magnetic flux transmitted by the injection antenna. This magnetic flux is proportional to the current sent into the injection solenoid. Thus, the electromagnetic glitch that is transmitted to the circuit depends on the current's variations. We also observed that sending longer pulses reduces the stress applied to the circuit.

Pulse voltage: To evaluate the influence of the pulse voltage, the test program has been set to a single LDR assembly instruction. `LDR R_o, [R_i, #offset]` loads the value pointed by `R_i` with offset `#offset` into the register `R_o`. For the test instruction, the register `R_i` pointed to a Flash memory address. To perform an analysis of the impact of the pulse's voltage, we needed to fix a suitable configuration for the other parameters. Those other parameters were set to some fixed values: we chose a configuration in which a fault occurs on the loaded value. For this experiment, the tested instruction was `LDR R4, [PC, #44]`. The initial value of R4 was 0x0 and PC+44 was a Flash memory address which contained 0x12345678. Since metastability

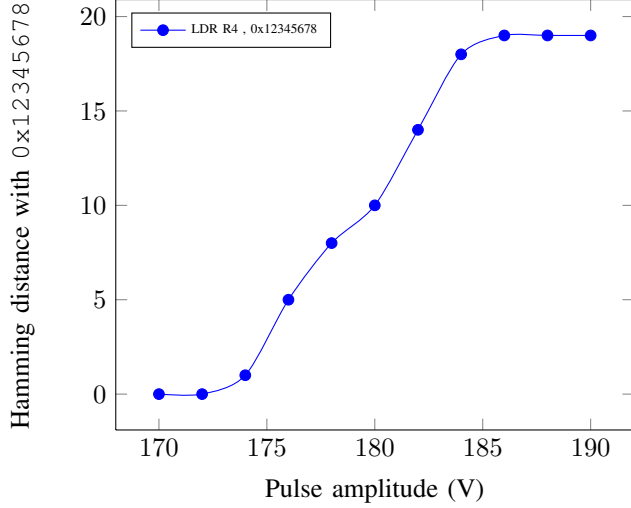


Figure 5: Hamming distance with 0x12345678 versus pulse's voltage

phenomena appear, for this experiment we take into account the faulty output with the highest occurrence rate. Table III shows the value in R4 for different values of the pulse voltage. According to those results, increasing the pulse voltage increases the Hamming weight of the loaded value. This pattern is highlighted by Fig. 5, which shows the Hamming distance with the 0x12345678 expected value versus the pulse's voltage. The same kind of trend has been obtained for different values for the probe position and the injection time. However, it seems that only instructions which loads a value from the Flash memory can lead to this kind of *set at 1* fault. Indeed, we did not manage to inject similar faults in case of a data transfer from the SRAM memory.

Table III: Influence of the pulse's voltage

Pulse voltage	Loaded value	Occurrence rate
170 V	1234 5678 (no fault)	100%
172 V	1234 5678 (no fault)	100%
174 V	9234 5678	73%
176 V	FE34 5678	30%
178 V	FFF4 5678	53%
180 V	FFFD 5678	50%
182 V	FFFF 7F78	46%
184 V	FFFF FFFB	40%
186 V	FFFF FFFF	100%
188 V	FFFF FFFF	100%
190 V	FFFF FFFF	100%

4) *Type of the executed instructions:* Our experiments highlighted a significant trend: we managed to inject faults on different types of instructions such as branch instructions, ALU instructions or load-store instructions. However, load instructions from the Flash memory were significantly easier to fault. The microcontroller we use has a Harvard

architecture. Every instruction fetch uses the instruction bus. Moreover, load instructions also use the data bus in the decode pipeline phase. As a consequence, section IV provides a more detailed study of the consequences of this fault injection in two cases: one case to highlight a fault on the instruction bus, another one to highlight a fault on the data bus. On the one hand, we study the effects on a generic single instruction. On the other hand, we study the specific case of a load instruction from the Flash memory.

IV. EXPERIMENTS ON THE DATA AND INSTRUCTION BUS

The two following subsections detail the results we obtained when trying to inject faults into the control flow or the data flow of the target program. In order to minimize the side effects which may happen when studying a big number of assembly instructions, the following results have been obtained for two classes of test applications. To highlight faults on the control flow, we use a sequence of NOP instructions [26]. Since NOP instructions have no effect, a faulty output will be easier to notice and to explain. To highlight faults on the data flow, the test application we use is a single LDR instruction which loads data from memory into a register. The initial values at the beginning of our target function are detailed in Table IV. Those beginning values are the same for the two following experiments. The comparison between the initial values, the output ones and the expected ones helps us to have a better understanding of possible instruction replacements effects.

Table IV: Initial values at the beginning of the execution

Piece of data	Value
r0	A memory address in RAM
r1 to r4	0x1 to 0x4
r5 and r6	Not relevant
r7	0x100
r8 to r12	0x00
Address pointed by r0	0x00

A. Faults on the program flow

Faults on the program flow can be observed through instruction replacement faults thanks to the simulation. However, studying instruction replacement with two possible instruction sizes is a very tough task. Since every fetch from the code memory is 32-bit wide, we need to consider several instruction replacement scenarios. With this approach, we can simulate the replacement of a 16-bit or 32-bit instruction by another 16-bit or 32-bit instruction. However, two 16-bit instructions might be replaced by two different 16-bit instructions. Similarly, a 32-bit instruction might be replaced by two 16-bit instructions. Those two cases would imply performing an almost-exhaustive search over 32 bits, which is not practical in our case. Though, we could partially bypass this problem by recording the number of clock cycles in our experiments. However, guessing the number of

executed instructions from the clock cycle count is not an easy task because of the complex instruction set. Observing this clock cycle count could theoretically enable us to exclude some replacement scenarios in further experiments. To highlight the possibility to inject faults on the program flow, the following experiment targeted a NOP sled. Since different position probes and different injection times lead to different results, the following results have been found for different experimental configurations of these parameters.

1) *Hardware exceptions:* Our fault injection sometimes led to an exception triggering. However, only Usage Fault exceptions were observed. More precisely, the No coprocessor exception and the Undefined instruction exception where the only subclasses of Usage Fault which could be observed. Both of these exceptions happen in the case of an invalid opcode. A possible explanation would be that a fault has been injected during the *fetch* or *decode* pipeline phases.

2) *Memory address:* In the initial state before the target instruction, `r0` points to a memory address in SRAM. The value of `r0` has been observed at this address instead of the expected value. For this particular case, instruction replacement simulation showed that the only possible instruction replacement is `STR r0, [r0, #0]`, which stores the value of `r0` at the address pointed by `r0` without any offset. Moreover, the value `0x100` has also been observed for another configuration. It turns out that `0x100` is also the value in `r7`.

3) *Other faults:* We also obtained faults on the general-purpose register `r7` and the program counter `r15`. These faults can also be explained by at least one assembly instruction replacement.

4) *Summary:* Obviously, the previous paragraphs do not aim at providing a complete list of the possible faults. Because of the huge number of possible configurations for the injection parameters, computing fault occurrence percentages would not be relevant. Nevertheless, these paragraphs highlight the fact that very few fault patterns were observed. We never got any fault on `r1-r6` and `r8-r14`. In an informal way, faults on `r7` and `r15` (pc) appeared much more often than faults on the memory address pointed by `r0`. Most of the 16-bit instructions can only manipulate the registers `r0` to `r7`. For example, a `MOVS r7, #FF` operation is assembled into a 16-bit instruction, while a `MOVS r8, #FF` is assembled into a 32-bit instruction. In a 16-bit instruction, `r7` is encoded by a 111 binary sequence. The fact that registers `r0-r6` are encoded with a smaller number of 1 in their encoding slot might explain this higher fault occurrence rate on the `r7` register. Similarly, branch instructions have many 1 in their slot. As a conclusion for this set of experiments, every faulty result we observed has at least one instruction replacement which can explain it. The first intuition of a *set at 1* fault model we saw for data fetches leads us to a more detailed analysis of the pipeline

stages in section V.

B. Faults on the data flow

For this experiment, we targeted a single `LDR r4, [PC, #44]` instruction. The initial value of `R4` was `0x0` and `PC+44` was a Flash memory address which contained `0x12345678` (this experiment used the same configuration as the one we had defined in III-B3). We obtained several faulty outputs such as `0xFE345678` or `0xFFFF45678`. We consider that every fault which cannot be explained by an instruction replacement is a fault on the data flow. In this experiment, the target `LDR r4, [PC, #44]` is a 16-bit instruction, followed by a 16-bit NOP instruction. The Thumb2 instruction set can only output a limited set of constants in a single data-processing instruction [24]. Thus, some of the faulty output values we observe, such as `0xFFFF45678`, could theoretically only be loaded with a single load from indirect register. Since the whole memory does not contain any `FFF4` pattern, a single load instruction could not explain this result. We performed an exhaustive search over the 16-bit and 32-bit instructions. No single instruction can lead to a result of `0xFFFF45678`. However, fault injection might have had an impact on two 16-bit instructions. To handle this issue, we performed another experiment, in which the target instruction was a `LDR r8, [PC, #44]`, with `0x12345678` stored at the address `PC+44`. Using `r8` instead of `r4` makes this instruction be assembled as a 32-bit instruction. Except the stack manipulation instructions, no 16-bit instruction can write a value into registers between `r8` and `r12`. For this new configuration, we were able to obtain several faulty values, such as `0xFFFF45679` or `0xFFFC5679`. With an exhaustive simulation, we can now guarantee that no single instruction can lead to such a result. Since a part of the faulty value is similar to the expected one, we can assume this fault injection had an impact on the data flow.

C. Analysis at a lower abstraction level

Underpowering a circuit or overclocking it leads to the same kind of timing violation faults [15], but knowing which among the clock tree or the power grid has been faulted is a tough task. To the best of our knowledge, recent research papers such as [27] claimed that the coupling between the injection probe and the circuit lies mainly in the power distribution network. According to the experiments from the previous section, electromagnetic glitch fault injection seems to enable us to perform attacks whose effect is equivalent to voltage or clock glitches, with a local effect that enables us to target either the instruction bus or the data bus. The following section deeply studies the bus transfers and provides an explanation for the faults we observed at a register-transfer level.

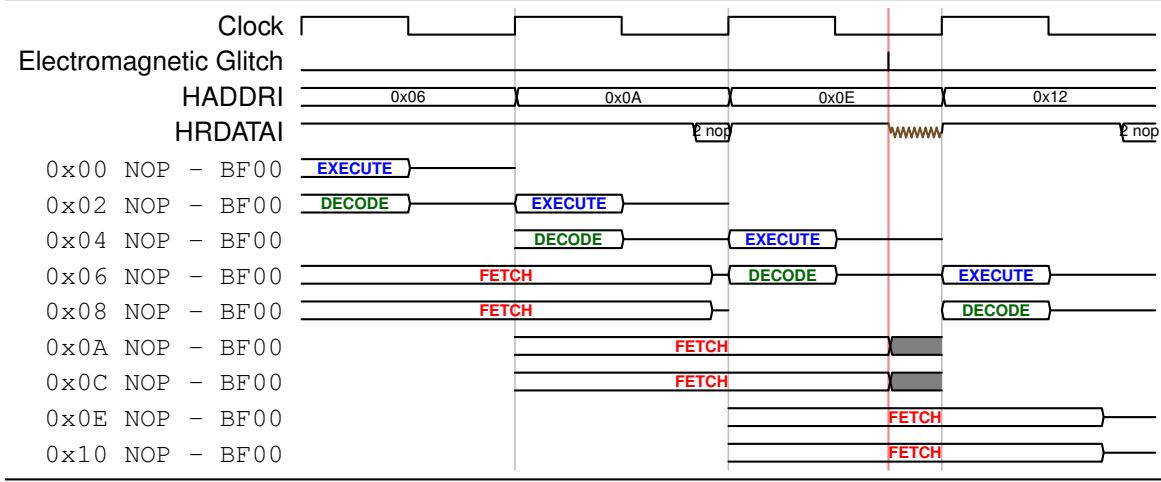


Figure 6: Bus transfers on the AHB bus for instruction memory

V. REGISTER-TRANSFER LEVEL FAULT MODEL

The results presented in section IV lead us to the basics of a definition of a fault model at the assembly level. By using this fault injection technique, an attacker can inject faults in two ways: modify the instruction to be executed or modify a data value in the case of a load instruction.

Our experiments also highlight another trend: we only managed to inject faults on data and instruction transfers from the Flash memory. The Flash memory has a slower response time than the SRAM memory. The *fetch* pipeline phase always requires a transfer from the instruction memory [23]. The operand fetch operation is performed during the *decode* phase. For load instructions, the *decode* phase requires a transfer from the data memory.

The microcontroller we use is based on a modified Harvard architecture, with separate buses for instruction and data. The buses are 32-bit wide and use the AMBA AHB-Lite structure [28]. Since electromagnetic glitch injection creates timing faults [17], we propose an explanation of the experimental faults we obtained based on a bus transfer analysis.

A. Instruction fetches

Fetching a piece of data or an instruction from the memory (either SRAM or Flash) requires at least two clock cycles. Fig 6 shows a chronogram of the AHB bus transfers when executing the target program. In this case, the target program is a NOP sled. Since instruction fetches are 32-bit wide, two 16-bit NOP instructions are fetched at each execution of the *fetch* pipeline stage. In the case of instruction which do not require an operand fetch, the *decode* and *execute* pipeline stages require at most half a clock cycle. The *fetch* stage requires one clock cycle during which the instruction address is written on the HADDRI bus. It also requires an extra clock cycle in which 32 bits from

the instruction memory are written on the HRDATAI bus [28]. In the event of a transfer from the SRAM memory, the values are written on the HRDATAI bus at the beginning of this extra clock cycle. Since the Flash memory has a longer response time, this value is written on the bus at the end of this clock cycle for a Flash transfer. In this situation, since electromagnetic fault injection leads to timing faults [17], the critical path appears to be this HRDATAI bus transfer.

Table V: Binary encoding of NOP and STR r0, [r0, #0]

Mnemonic	Inst.	Binary instruction	Hamming w.
NOP	BF00	10111111 00000000	7
STR r0, [r0, #0]	6000	01100000 00000000	2

We now consider the result presented in IV-A, in which a NOP is replaced by a STR r0, [r0, #0]. The binary encodings for the NOP and STR r0, [r0, #0] instructions are presented in Table V. As seen for an attack which targets the value loaded by a load instruction, it seems that the higher the stress we apply, the higher the fetched word's Hamming weight is. However, the situation seems different for instruction fetches. The fault models seems more complex than the *set at 1* model we had seen for data fetches.

The bus precharge values are not specified in the AHB bus intellectual property. They are chosen by the circuit's manufacturer. For the microcontroller we use, the HRDATAI bus does not seem to be precharged at 1, since a NOP instruction with a Hamming weight of 7 has been replaced by another instruction whose Hamming weight is 2. Moreover, since there must be some skew on this bus, some metastability phenomena (as presented in III-A) also appear. Considering this single example, a possible precharge value would be 0, or the microcontroller might use a more complex precharge strategy. For the moment, we are not able to infer more details about a possible HRDATAI bus precharge.

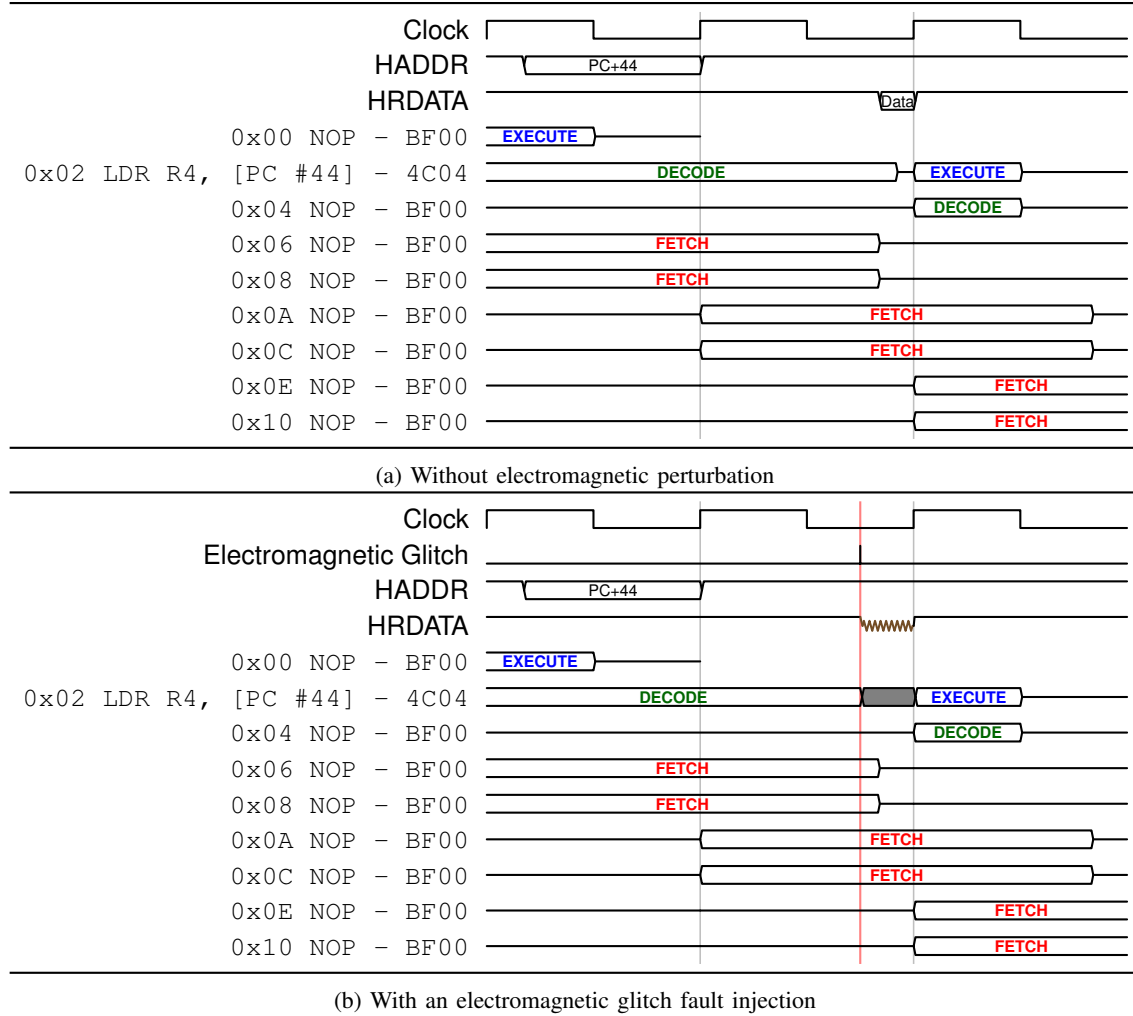


Figure 7: Bus transfers on the AHB bus for data memory

To sum up, in the case of a bus transfer from the Flash memory, the critical path that is faulted by an electromagnetic fault injection seems to be the HRDATAI bus transfer. Thus, this fault injection can target any instruction fetch from the Flash memory, which potentially makes this attack scenario very harmful.

B. Data fetches

The situation for data fetches is very similar to the one we describe for instruction fetches. Since electromagnetic fault injection has a local effect, it is possible to find a probe position where we only inject faults on the data bus and do not reach the instruction bus. For this attack scenario, the critical path seems to be the HRDATA bus transfer. Fig. 7a shows data bus transfers in the case of a single LDR instruction (similar to the previous experiments) without fault injection. Fig. 7b shows the same bus transfers in the case of a fault injection.

Metastability phenomena also appear for this fault injection, but the global trend corresponds to a *set at 1* fault model. This value depends on the microcontroller's bus precharge strategy, which is specific to each implementation. This trend enables us to define a more precise fault model for data fetches, in which the attacker can bring the loaded value closer to the value of the bus precharge.

VI. RELATED WORKS

This section outlines some research papers that are related to the study we presented in this paper. These papers are grouped into three subcategories: electromagnetic fault injection techniques, fault models on microcontrollers and proposed countermeasures against a given fault model.

1) *Electromagnetic fault injection:* In [17], Dehbaoui *et al.* do a practical fault injection on a software implementation of the AES algorithm by using electromagnetic glitches. In [25], Carrier performs a study of the effects of electromagnetic fault injection on two microcontrollers at

an electric level. His work mostly explains the influence of several parameters related to the coil. He also studies the influence of the injection time. However, his study does not focus on the faults that were produced.

2) *Fault models on microcontrollers:* In [29], Barenghi *et al.* study the effects on low-voltage fault attacks on an ARM9 microprocessor. They describe several effects on loads from memory or on instruction replacement. In [13], Balasch *et al.* present a black-box approach which is quite similar to the one proposed in this paper. They use clock glitches as a fault injection mean and perform their experiment on a 8-bit microcontroller. We also use the same kind of in-depth analysis. However, their study is performed on a very different architecture with a different bus precharge configuration. We also automated the instruction replacement search by performing an exhaustive instruction replacement simulation over the instruction set. In [26], Spruyt proposes an approach whose aim is to define a generic method on how to build a fault model for microcontrollers. His situation is also quite similar to the one presented in this paper. Since having access to some internal data on a real microcontroller may be hard, he proposes a way to obtain information about the induced faults by analyzing the faulty outputs of different groups of instructions. Several articles have been published in which the authors assume an attacker can skip or replace an instruction by another one [18] [30]. For example, in [31], Berzati *et al.* suppose an attacker can replace an addition instruction by an exclusive or instruction.

3) *Countermeasures:* Several countermeasures schemes have been defined to protect embedded processor architectures against specific fault models. All those countermeasure schemes might be reinforced by studies similar to the one presented in this paper, which could provide a more precise knowledge about the fault model. At a hardware level, [32] proposes to use integrity checks to ensure that no instruction replacement took place. Nevertheless, many countermeasures to protect assembly code without modifying the microcontroller's architecture have been defined. In [21], Barenghi *et al.* propose three countermeasure schemes based on instruction duplication, instruction triplication and parity checking. Their countermeasures enable different levels of fault detection and correction against instruction skips or some instruction modifications. In [33], Medwed *et al.* propose a generic approach based on the use of specific algebraic structures named $AN+B$ codes. Their approach enables to protect both the control and data flow. An application to an AES implementation has also been detailed in [34].

VII. CONCLUSION

We have presented a detailed study about the effects of an electromagnetic glitch fault injection on a state-of-the-art microcontroller. However, working with a real microcontroller

in a black-box approach creates several constraints when trying to build a practical experimental process. Because of the lack of details about the microcontroller's design and architecture, we have proposed this top-down approach which aims at building a suitable lower-level explanation for the faults we observed at an assembly level. Moreover, we also lack information about the bus precharge strategy on the microcontroller we use. Future experiments will try to use more advanced debug techniques in order to get more accurate information about the executed instructions.

Finally, we do not claim this register-transfer level hypothesis is the only reason why faults appear at an assembly level. This paper aims at providing a first understanding of the faults an electromagnetic glitch fault injection can induce on an embedded program. And the lower-level model we propose could explain all the previous experimental results we obtained. Furthermore, this fault model looks very similar to the ones which can be found in previous works about clock or voltage glitches. Hence, electromagnetic glitches seem to induce timing constraints violations on the bus transfers from the Flash memory. Thus, on a standard circuit, electromagnetic fault injection could enable an attacker to bypass some countermeasures against traditional timing fault injection means such as clock or voltage glitches.

These experiments confirm the fact that an attacker could change an instruction into another one and change the value of a piece of data loaded from the Flash memory. But they also provide a more accurate fault model, in which some instructions or registers seem to be more vulnerable than others. On this architecture, faults on the data flow lead to an increased Hamming weight on the loaded piece of data. This behaviour highly depends on the microcontroller's bus precharge strategy. These observations can lead to the definition of an assembly-level fault model, and enable to build more specific and accurate countermeasures. These ideas will be studied more precisely in future works.

REFERENCES

- [1] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *Advances in Cryptology - CRYPTO'96*, pp. 104–113, 1996. [Online]. Available: <http://www.springerlink.com/index/4el17cvre3gxt4gd.pdf>
- [2] D. Agrawal, B. Archambeault, J. R. Rao, P. Rohatgi, and Y. Heights, "The EM Side-Channel(s)," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science, B. S. Kaliski, c. K. Koç, and C. Paar, Eds., vol. 2523. Berlin, Heidelberg: Springer Berlin Heidelberg, Feb. 2003, pp. 29–45. [Online]. Available: <http://www.springerlink.com/index/10.1007/3-540-36400-5>
- [3] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference*, Santa Barbara, California, USA, 1999, pp. 1–10. [Online]. Available: <http://www.springerlink.com/index/kx35ub53vtrkh2nx.pdf>

- [4] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, "Simple Photonic Emission Analysis of AES Photonic Side Channel Analysis for the Rest of Us," *Cryptographic Hardware and Embedded Systems - CHES 2012*, pp. 41–57, 2012.
- [5] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, vol. 1233, pp. 37–51, 1997. [Online]. Available: <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.9764>
- [6] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb. 2006. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1580506>
- [7] D. Karaklajić, J.-M. Schmidt, and I. Verbauwhede, "Hardware Designer's Guide to Fault Attacks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 258754, pp. 1–1, 2013. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6425517<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6425517>
- [8] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov. 2012. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6178001
- [9] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," *Cryptographic Hardware and Embedded Systems - CHES 2002*, vol. 2523, no. August, pp. 2–12, 2003. [Online]. Available: <http://www.springerlink.com/index/dmjmf1pt7lr1c962.pdf>
- [10] J.-M. Schmidt and M. Hutter, "Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results," in *Proceedings of the 15th Austrian Workshop on Microelectronics - Austrochip 2007*, Graz, Austria, 2007. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.5741&rep=rep1&type=pdf>
- [11] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin, "Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA," *Journal of Cryptology*, vol. 24, no. 2, pp. 247–268, Oct. 2010. [Online]. Available: <http://www.springerlink.com/index/10.1007/s00145-010-9083-9>
- [12] M. Agoyan, J.-m. Dutertre, A.-p. Mirbaha, D. Naccache, A.-l. Ribotta, and A. Tria, "How to Flip a Bit?" in *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*. IEEE, 2010, pp. 235–239.
- [13] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Sep. 2011, pp. 105–114. [Online]. Available: <http://www.cosic.esat.kuleuven.be/publications/article-2059.pdf>
- [14] J. J. A. Fournier, S. Moore, H. Li, R. Mullins, and G. Taylor, "Security Evaluation of Asynchronous Circuits," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, 2003, pp. 137–151.
- [15] L. Zussa, J.-m. Dutertre, J. Clédière, B. Robisson, and A. Tria, "Investigation of timing constraints violation as a fault injection means," in *27th Conference on Design of Circuits and Integrated Systems (DCIS)*, Avignon, France, 2012. [Online]. Available: <http://hal-emse.ccsd.cnrs.fr/emse-00742652/>
- [16] S. Skorobogatov, "Local Heating Attacks on Flash Memory Devices," in *IEEE International Workshop on Hardware-Oriented Security and Trust, 2009 - HOST'09*. IEEE, 2009, pp. 1–6. [Online]. Available: http://www.cl.cam.ac.uk/~sps32/host2009-flash_heat.pdf
- [17] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES," *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 7–15, Sep. 2012. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6305224>
- [18] J.-M. Schmidt and C. Herbst, "A Practical Fault Attack on Square and Multiply," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J. P. Seifert, Eds. IEEE, Aug. 2008, pp. 53–58. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4599557>
- [19] S. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *Computers, IEEE Transactions on*, vol. 49, no. September 1996, pp. 967–970, 2000. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=869328
- [20] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Proceedings of the 17th Annual International Cryptology Conference*, no. September 1996, Santa Barbara, California, USA, 1997. [Online]. Available: <http://info.psu.edu.sa/psu/cis/abuelyaman/Research/DFA-Secret-Key.pdf>
- [21] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni, "Countermeasures against fault attacks on software implemented AES," in *Proceedings of the 5th Workshop on Embedded Systems Security - WESS '10*. New York, New York, USA: ACM Press, 2010, pp. 1–10. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1873555>
- [22] I. Verbauwhede, D. Karaklajić, and J.-M. Schmidt, "The Fault Attack Jungle - A Classification Model to Guide You," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Sep. 2011, pp. 3–8. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6076462>
- [23] J. Yiu, *The Definitive Guide To The ARM Cortex-M3*. Elsevier Science, 2009.
- [24] ARM, "ARM Architecture Reference Manual - Thumb-2 Supplement," 2005.

- [25] S. Carlier, "Electro Magnetic Fault Injection," University of Amsterdam, Amsterdam, Tech. Rep., 2012. [Online]. Available: <http://staff.science.uva.nl/~delaat/rp/2011-2012/p19/report.pdf>
- [26] A. Spruyt, "Building fault models for microcontrollers," University of Amsterdam, Amsterdam, Tech. Rep., 2012. [Online]. Available: <http://staff.science.uva.nl/~delaat/rp/2011-2012/p61/report.pdf>
- [27] F. Poucheret, L. Chusseau, B. Robisson, and P. Maurine, "Local electromagnetic coupling with CMOS integrated circuits," in *2011 8th Workshop on Electromagnetic Compatibility of Integrated Circuits*. IEEE, 2011, pp. 137–141.
- [28] ARM, "AMBA 3 AHB-Lite Protocol," 2006.
- [29] A. Barengi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low Voltage Fault Attacks on the RSA Cryptosystem," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Sep. 2009, pp. 23–31. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5412860>
- [30] J.-M. Schmidt and M. Medwed, "A Fault Attack on ECDSA," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Sep. 2009, pp. 93–99. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5412852>
- [31] A. Berzati, C. Canovas-Dumas, and L. Goubin, "Fault analysis of Rabbit: toward a secret key leakage," *Progress in Cryptology - INDOCRYPT 2009*, pp. 72–87, 2009. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-10628-6_5
- [32] M. H. Nguyen, B. Robisson, M. Agoyan, and N. Drach, "Low-cost recovery for the code integrity protection in secure embedded processors," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, Jun. 2011, pp. 99–104. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5955004>
- [33] M. Medwed and J.-M. Schmidt, "A Generic Fault Countermeasure Providing Data and Program Flow Integrity," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Aug. 2008, pp. 68–73. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4599559>
- [34] M. Medwed, "A Continuous Fault Countermeasure for AES Providing a Constant Error Detection Rate," in *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Aug. 2010, pp. 66–71. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5577364>